

Boiler Plating Database Resource Cleanup

Part I

Paul Grenyer

I've been using Java for nearly twelve months now and I am finding that I like it. There are only two things that I have discovered so far that make me wonder what the creators of Java were thinking: exception handling and layout managers. I'll cover layout managers in a later article.

Java is a garbage collected language. Which means that, most of the time, you don't have to worry how the memory previously used by dead objects is cleaned up. To me garbage collection has always felt a bit like a knee jerk reaction for people who can't use smart pointers properly in C++ and C programmers who must pay very close attention to the points at which they release memory allocated to the heap. If garbage collection is meant to help you clean up memory, why hasn't something been developed to help objects release resources? Java has finalizers, but as Joshua Bloch points out in item 7 of Effective Java [EffectiveJava] finalizers cannot be relied upon (see sidebar 1). Proper cleanup of resources is left to the developer whose only real friend is `finally`.

In languages like C# the `IDisposable` [IDisposable] interface can be used to aid cleanup. It has a single method called `Dispose`. All of an object's cleanup code should be placed in this method and calling the method manually or via `using` ensures resources are released. The draw back is that even `IDisposable` requires the client code to use it. There is nothing like this available in Java, but I think I may have an even better solution which takes away the dependence on a classes user to do any cleanup.

Tony Barrett-Powell in “Handling Exceptions In Finally” [HEiF] and Alan Griffiths in “Exceptional Java” [ExceptionalJava] and “More Exceptional Java” [MoreExceptionalJava] both discuss methods of dealing with database related cleanup and exceptions, but their solutions are still verbose and can be boiler plated. These articles form the basis of the boiler plate I present here.

In this article I'll look briefly at the vast amount of exception handling code that must be written to cleanly and efficiently close result set, statement and connection objects, without relying on finalizers when accessing a database and then more in depth at one possible method of boiler plating it.

The Problem

The problem is simple. Cleaning up after querying a database in Java is unnecessarily verbose and complex. Plain and simple. I'll start with an example that demonstrates the problem. The system I'm working on currently uses a number of web services. We have a set of web services on the production box, another on the development box and another on our local machines. The system asks the relevant database for the location of the web service based on the services' name:

```
try
{
    Class.forName(driver);
    Connection con = DriverManager.getConnection(
        connectionString,
```

```

        username,
        password );
    try
    {
        PreparedStatement ps = con.prepareStatement(
            "select url from services where name = 'Instruments'");

        try
        {
            ResultSet rs = ps.executeQuery();
            if(rs.next())
            {
                System.out.println(rs.getString("url"));
            }

            try
            {
                rs.close();
            }
            catch(SQLException e)
            {
                // Report Error
            }
        }
        finally
        {
            try
            {
                ps.close();
            }
            catch(SQLException e)
            {
                // Report Error
            }
        }
    }
    finally
    {
        try
        {
            con.close();
        }
        catch(SQLException e)
        {
            // Report Error
        }
    }
}
catch(Exception e)
{
    // Report Error
}

```

This is a lot of code to get one string out of a database and most of it must be repeated every time a database is accessed. Most of it is error handling and resource management. In fact I've over simplified it. For a discussion of how error handling and resource management should really be handled see "Handling Exceptions in Finally" mentioned above.

The Sun Java Documentaton [JavaDocs] states the following for the Connection interface's close method:

Releases this `Connection` object's database and JDBC resources immediately instead of waiting for them to be automatically released.

Note: A `Connection` object is automatically closed when it is garbage collected. Certain fatal errors also close a `Connection` object.

It also states the following for the `Statement` interface's `close` method:

Releases this `Statement` object's database and JDBC resources immediately instead of waiting for this to happen when it is automatically closed. It is generally good practice to release resources as soon as you are finished with them to avoid tying up database resources.

Note: A `Statement` object is automatically closed when it is garbage collected. When a `Statement` object is closed, its current `ResultSet` object, if one exists, is also closed.

And the following for the `ResultSet` interface's `close` method:

Releases this `ResultSet` object's database and JDBC resources immediately instead of waiting for this to happen when it is automatically closed.

Note: A `ResultSet` object is automatically closed by the `Statement` object that generated it when that `Statement` object is closed, re-executed, or is used to retrieve the next result from a sequence of multiple results. A `ResultSet` object is also automatically closed when it is garbage collected.

This can all be interpreted in a number of two ways. The first is that everything gets closed when it is garbage collected, so there is no need to explicitly close anything. This relies on the appropriate finalizers getting called but, as Bloch tells us, Java provides no guarantee that a finalizer will ever be called, even when an object is garbage collected.

Another method is to explicitly close `Statement` and `Connection` objects as `Statement` objects clean up their associated `ResultSet` objects. Drawbacks include any error caused by closing the `ResultSet` is potentially ignored and the resource is not released as soon as it could be.

Yet another method is to explicitly close everything. This is the most code, but releases resources and handles any error as soon as a resource is finished with, making it most efficient way of using resources.

I favour the third and final method. It is more code, but boiler plating will mean most of it only need to be written once. Everything will be cleaned up as soon as possible, all errors can be trapped and reported and nothing is left to chance.

The `close` methods for `Connection`, `Statement` and `ResultSet` objects can all throw if there is an exceptional circumstance. In Item 65 of Effective Java (see sidebar) Bloch explains that exceptions should not be ignored. He also describes certain circumstances where it *might* be ok to ignore or log these types of exceptions. The `Connection`, `Statement` and `ResultSet` `close` methods could be considered one of these situations.

A Solution

In Another Tale of Two Patterns [AToTP] Kevlin Henney describes the Finally For Each Release and Execute Around Method (EAM) patterns. Both are applicable to the problem. Here I will look at the Finally For Each Release solution and in Part II a solution using Execute Around Method.

In its basic for the Finally After Each Release pattern looks like this:

```
resource.acquire();
try
{
    resource.use();
}
finally
{
    resource.release();
}
```

which describes accurately how database resources should be created, used and cleaned up.

Connection Policies

Let's start by looking at database connection management. There are a number of ways of creating and using a database connection. Probably the two most common are:

- Creating it with a driver (e.g. JDBC) and a connection string.
- Using an existing connection created and cleaned up somewhere else.

Sensible boiler plate code should support both of these methods and allow custom creation of database connections, which makes using a policy ideal:

```
public interface ConnectionPolicy
{
    abstract Connection connect()
        throws ConnectionException;

    abstract void disconnect(final Connection con)
        throws ConnectionException;
}
```

All the boiler plate code has to do is call `connect` to get a `Connection` object and `disconnect` to release it when it's done. It does not need to care how the object is created or how, or even if, it is released. This means that `ConnectionPolicy` could even be implemented as a `Connection` pool. The implementation of `ConnectionException` is trivial and simply there to force a common exception type to be thrown by the policy. It extends `ResourceHandlerException`, which we'll look at later:

```
public class ConnectionException extends ResourceHandlerException
{
    public ConnectionException( final String message,
                               final Throwable throwable)
    {
        super(message, throwable);
    }
}
```

Most connections, if they're closed by the policy, will be closed in the same way: by calling `close` on the `Connection` object. So it's worth putting the common close code into an abstract class:

```
public abstract class AbstractConnectionPolicy
    implements ConnectionPolicy
{
    @Override
    public void disconnect(final Connection con)
        throws ConnectionException
    {
        try
        {
            if (con != null)
            {
                con.close();
            }
        }
        catch (final SQLException e)
        {
            throw new ConnectionException(e.getMessage(), e);
        }
    }
}
```

All this does is check that the `Connection` object is not null, call `close` on it and catch and translate any exception it might throw. Implementing a connection policy for an existing connection is then very simple:

```
public class ExistingConnection extends AbstractConnectionPolicy
{
    private final Connection con;
    private final boolean cleanup;

    public ExistingConnection(final Connection con, boolean cleanup)
    {
        this.con = con;
        this.cleanup = cleanup;
    }

    public ExistingConnection(final Connection con)
    {
        this(con, false);
    }

    @Override
    public Connection connect() throws ConnectionException
    {
        return con;
    }

    @Override
    public void disconnect(final Connection con)
        throws ConnectionException
    {
        if (cleanup)
        {
            super.disconnect(con);
        }
    }
}
```

A Connection object is passed in through the constructor and passed back via connect. A overloaded constructor allows the cleanup flag to be set, by default it is not set. When disconnect is called the flag is checked to see if the connection should be cleaned up. Although it's simple, it's not as simple as it could be as I have added the cleanup flag, but the policy is more flexible this way.

A policy for creating a connection from a connection string and driver is a little more involved, but the beauty of boiler plate code is that you only have to write it once:

```
public class StringConnection extends AbstractConnectionPolicy
{
    private final String driver;
    private final String connectionString;
    private String database = null;
    private String username = null;
    private String password = null;

    public StringConnection(        final String driver,
                                final String connectionString)
    {
        this.driver = driver;
        this.connectionString = connectionString;
    }

    public StringConnection setUser(    final String username,
                                        final String password)
    {
        this.username = username;
        this.password = password;
        return this;
    }

    public StringConnection setDatabase(final String database)
    {
        this.database = database;
        return this;
    }

    @Override
    public Connection connect() throws ConnectionException
    {
        Connection con = null;

        try
        {
            Class.forName(driver);
            con = DriverManager.getConnection( connectionString,
                                            username,
                                            password);

            useDatabase(con, database);
        }
        catch(ClassNotFoundException e)
        {
            throw new ConnectionException(e.getMessage(), e);
        }
        catch(SQLException e)
        {
            throw new ConnectionException(e.getMessage(), e);
        }

        return con;
    }
}
```

```

private void useDatabase(    final Connection con,
                           final String database)
                           throws SQLException
{
    if (database != null)
    {
        try
        {
            final Statement stmt = con.createStatement();

            try
            {
                final StringBuilder sql
                    = new StringBuilder("USE ");
                sql.append(database);
                sql.append(" ");
                stmt.execute(sql.toString());
            }
            finally
            {
                stmt.close();
            }
        }
        catch(final SQLException e1)
        {
            try
            {
                con.close();
            }
            catch(final SQLException e2)
            {
                // Swallow
            }

            throw e1;
        }
    }
}

```

StringConnection extends AbstractConnectionPolicy and takes advantage of the common disconnect method. The constructor takes a driver string (e.g. "sun.jdbc.odbc.JdbcOdbcDriver") and a connection string (e.g. "jdbc:odbc:Marauder") and uses them to set the appropriate members. The setUser and setDatabase methods are a variation on the builder pattern, as described by Joshua Bloch's second Effective Java item (see sidebar 2), that allows a username and password and a default database to be set optionally. For example:

```

final ConnectionPolicy conPolicy
    = new StringConnection(driver, connectionString)
        .setUser(username, password)
        .setDatabase(database);

```

The connect override uses the driver and connectionString objects to create a Connection object and catches, translates and rethrows any exceptions. It also calls the useDatabase method. If the database object has been set it uses it to set the current database by building the appropriate SQL statement and using a Statement object to execute it. The Statement object is cleaned up by a finally block. If an exception is thrown at any point, the

Connection object is closed, but any close exception ignored in favour of the original exception and then the exception rethrown. This is one of those rare cases where an exception can be ignored, but should be logged.

Resource Handler

With the Finally For Each Release pattern and the `ConnectionPolicy` interface as a starting point it is possible to start building up a class that will handle the cleanup of resources automatically:

```
public class ResourceHandler<Value>
{
    private final ConnectionPolicy conPolicy;

    public ResourceHandler(final ConnectionPolicy conPolicy)
    {
        this.conPolicy = conPolicy;
    }

    public Value executeQuery(final String sql)
        throws ResourceHandlerException
    {
        Value result = null;

        Connection con = conPolicy.connect();

        try
        {
            // use
        }
        catch (final SQLException e)
        {
            // report error
        }
        finally
        {
            try
            {
                conPolicy.disconnect(con);
            }
            catch (final ConnectionException e)
            {
                // report error
            }
        }

        return result;
    }
}
```

The constructor takes and stores a `ConnectionPolicy` object. The `executeQuery` method uses the policy to create, use and cleanup a `Connection` object. A new feature called Generics [Generics] was introduced in Java 1.5. One of the advantages of generics is that you can specify the type used by a class when an instance of that class is declared. In the case of the `ResourceHandler` the return type of the `executeQuery` method is parametrized so that it can return any type. As we'll see later, this is useful if the SQL query is returning something other than a single string; such as a list of strings or key value pairs.

The next step after creating the connection is to create a Statement object:

```
public Value executeQuery(final String sql)
    throws ResourceHandlerException
{
    Value result = null;

    Connection con = conPolicy.connect();

    try
    {
        final Statement stmt = createStatement(con);

        try
        {
            // use statement
        }
        finally
        {
            try
            {
                stmt.close();
            }
            catch(final SQLException e)
            {
                // report error
            }
        }
    }
    catch(final SQLException e)
    {
        // report error
    }
    finally
    {
        try
        {
            conPolicy.disconnect(con);
        }
        catch(final ConnectionException e)
        {
            // report error
        }
    }

    return result;
}

protected Statement createStatement(final Connection con)
    throws SQLException
{
    return con.createStatement();
}
```

A Statement object is created by the `createStatement` method, which is protected so that it can be overridden in a subclass if a different type of statement needs to be created. A `ResultSet` object can be created in much the same way:

```
public Value executeQuery(final String sql)
    throws ResourceHandlerException
{
```



```

{
    Value result = null;

    Connection con = conPolicy.connect();

    try
    {
        final Statement stmt = createStatement(con);

        try
        {
            final ResultSet rs = getResultSet(stmt,sql);

            try
            {
                result = getValue(rs);
            }
            catch(SQLException e)
            {
                // report error
            }
            finally
            {
                try
                {
                    rs.close();
                }
                catch(final SQLException e)
                {
                    // report error
                }
            }
        }
        catch(final SQLException e)
        {
            // report error
        }
        finally
        {
            try
            {
                stmt.close();
            }
            catch(final SQLException e)
            {
                // report error
            }
        }
    }

    ...
}

protected Value getValue(final ResultSet rs) throws SQLException
{
    return null;
}

```

The return value will always be null unless `getValue` is overridden in a subclass. Finally a way of reporting errors needs to be implemented. In some circumstances it may be sufficient to send the error to standard out via `printStackTrace`. In others the throwing on an exception may be desirable. The use of a policy allows these and other error reporting methods to be implemented:

```

public interface ErrorPolicy
{
    abstract void handleError(final Exception e)
                        throws ResourceHandlerException;

    abstract void handleCloseError(final Exception e)
                        throws ResourceHandlerException;
}

```

There is a general error handler and handler for errors caused by releasing resources. because a client may wish to handle release errors differently, for example logging them rather than throwing.

The ResourceHandlerException looks like this:

```

public class ResourceHandlerException extends Exception
{
    public ResourceHandlerException(String message, Throwable throwable)
    {
        super(message, throwable);
    }
}

```

The default error policy ought to throw an exception and should only allow the first exception it receives to be thrown. This is so that the first exception detailing the real problem doesn't get lost when subsequent exceptions are thrown:

```

public class ThrowOnError implements ErrorPolicy
{
    private Exception exception = null;

    @Override
    public void handleError(Exception e) throws ResourceHandlerException
    {
        if (exception == null)
        {
            exception = e;
            throw new ResourceHandlerException(e.getMessage(), e);
        }
    }

    @Override
    public void handleCloseError(Exception e)
                        throws ResourceHandlerException
    {
        handleError(e);
    }
}

```

Error policies are integrated into ResourceHandler using another builder method and by calling handleError or handleCloseError in the appropriate catch blocks:

```

public class ResourceHandler<Value>
{
    private final ConnectionPolicy conPolicy;
    private ErrorPolicy errorPolicy = new ThrowOnError();

    ...

    public ResourceHandler<Value>

```

```

        setErrorPolicy(final ErrorPolicy errorPolicy)
    {
        this.errorPolicy = errorPolicy;
        return this;
    }

    public Value executeQuery(final String sql)
        throws ResourceHandlerException
    {
        Value result = null;

        Connection con = conPolicy.connect();

        try
        {
            final Statement stmt = createStatement(con);

            try
            {
                final ResultSet rs = getResultSet(stmt,sql);

                try
                {
                    result = getValue(rs);
                }
                catch(SQLException e)
                {
                    errorPolicy.handleError(e);
                }
                finally
                {
                    try
                    {
                        rs.close();
                    }
                    catch(final SQLException e)
                    {
                        errorPolicy.handleCloseError(e);
                    }
                }
            }
            catch(final SQLException e)
            {
                errorPolicy.handleError(e);
            }
            finally
            {
                try
                {
                    stmt.close();
                }
                catch(final SQLException e)
                {
                    errorPolicy.handleCloseError(e);
                }
            }
        }
        catch(final SQLException e)
        {
            errorPolicy.handleError(e);
        }
        finally
        {

```

```

        try
        {
            conPolicy.disconnect(con);
        }
        catch (final ConnectionException e)
        {
            errorPolicy.handleCloseError(e);
        }
    }

    return result;
}
...
}

```

Java has another wonderful feature called anonymous classes. In *Java In A Nutshell* [JavaInANutshell] David Flanagan describes anonymous classes as:

“...a local class without a name. Instead of defining a local class and then instantiating it, you can often use an anonymous class to combine these two steps”

An anonymous class can be used to implement a custom `getValue` method. The following example shows how anonymous classes can be used to execute a SQL statement that returns a single string:

```

final String url = new ResourceHandler<String>(
    new StringConnection(driver,connectionString))
{
    @Override
    protected String getValue(final ResultSet rs) throws SQLException
    {
        if (rs.next())
        {
            return rs.getString("url");
        }
        return null;
    }
}.executeQuery("select url from services where name = 'Instruments'");

```

Alternatively if multiple strings are required:

```

final List<String> urls = new ResourceHandler<List<String>>
    (new StringConnection(driver,connectionString))
{
    @Override
    protected List<String> getValue(final ResultSet rs)
        throws SQLException
    {
        List<String> result = new ArrayList<String>();

        while(rs.next())
        {
            result.add(rs.getString("url"));
        }
        return result;
    }
}.executeQuery("select url from services");

```

Executing Statements That Do Not Return a ResultSet

The `ResourceHandler` works fine until you do something like this:

```
new ResourceHandler(conPolicy).executeQuery(
    "insert into services ([Name],[url])
    values ('Engine','http://prodserv01/axis/services/Engine')");
```

The above statement doesn't return a result set and unhelpfully Java throws an exception to let you know:

```
ResourceHandlerException: No ResultSet was produced
```

The best way to get around this is to add an `execute` method, similar to the `executeQuery` method, but without a return value:

```
public void execute(final String sql) throws ResourceHandlerException
{
    Connection con = conPolicy.connect();

    try
    {
        final Statement stmt = createStatement(con);

        try
        {
            execute(stmt, sql);
        }
        catch (final SQLException e)
        {
            errorPolicy.handleError(e);
        }
        finally
        {
            try
            {
                stmt.close();
            }
            catch (final SQLException e)
            {
                errorPolicy.handleCloseError(e);
            }
        }
    }
    catch (final SQLException e)
    {
        errorPolicy.handleError(e);
    }
    finally
    {
        try
        {
            conPolicy.disconnect(con);
        }
        catch (final ConnectionException e)
        {
            errorPolicy.handleCloseError(e);
        }
    }
}
```

```

...

protected void execute(final Statement stmt, final String sql)
                        throws SQLException
{
    stmt.execute(sql);
}

```

Finally, if you need to execute multiple statements that do not return result sets, you can do this:

```

new ResourceHandler(conPolicy).execute("..");
new ResourceHandler(conPolicy).execute("..");
new ResourceHandler(conPolicy).execute("..");
new ResourceHandler(conPolicy).execute("..");
new ResourceHandler(conPolicy).execute("..");

```

However, with the `StringConnection` policy this would create and destroy the connection for each statement call. The alternative is modify `execute` to take an array of statements and iterate through them creating a `Statement` object for each:

```

public void execute(final String[] sql) throws ResourceHandlerException
{
    Connection con = conPolicy.connect();

    try
    {
        for(final String s : sql)
        {
            final Statement stmt = createStatement(con);

            try
            {
                execute(stmt,s);
            }
            catch(final SQLException e)
            {
                errorPolicy.handleError(e);
            }
            finally
            {
                try
                {
                    stmt.close();
                }
                catch(final SQLException e)
                {
                    errorPolicy.handleCloseError(e);
                }
            }
        }
    }
    catch(final SQLException e)
    {
        errorPolicy.handleError(e);
    }
    finally
    {
        try
        {
            conPolicy.disconnect(con);
        }
        catch(final ConnectionException e)

```

```

        {
            errorPolicy.handleCloseError(e);
        }
    }
}

```

This has the draw back that even if you have just a single statement to execute it has to be passed in as an element of an array. This can be easily overcome by adding an execute overload:

```

public void execute(final String sql) throws ResourceHandlerException
{
    execute(new String[]{sql});
}

```

Conclusion

I was reviewing some code where I work recently and I came across this:

```

...

finally
{
    try
    {
        if (rs != null)
        {
            rs.close();
        }
        if (stmt != null)
        {
            stmt.close();
        }
        if (con != null)
        {
            con.close();
        }
    }
    catch (SQLException e)
    {
        // Translate exception
    }
}

```

I summoned my two co-developers over, one of whom had written it, and asked them to find the problem. They looked at it for a good 90 seconds before I started offering clues. As soon as I asked what would happen if `rs.close()` threw the penny dropped and the lights went on. The answer of course is that the `Statement` and `Connection` objects would not get closed. The real break through came when one commented that Java programs must be failing to close database connections properly all over the place. The way Java handles resource cleanup is ludicrously verbose and the idea of a method throwing when a resource is closed just plain ridiculous.

However, I believe that I have shown here that this can be overcome using Finally For Each Release and the amount of code needed to query a database reduced significantly with use of some simple boiler plate.

In part II I will look at another possible solution using Execute Around Method.

Sidebar 1: Effective Java Item 7: - Avoid Finalizers

This item could have been written to put me straight. I'm the C++ programmer it speaks about and I was desperate for finalizers to be Java's destructor. They aren't. In fact, unlike C++'s destructors they are unpredictable, often dangerous and generally unnecessary. Ok, so if you're not careful with exceptions, destructors in C++ can be dangerous too. Java finalizers are worse.

The item explains that you should never do anything time critical in finalizers as the JVM is "tardy" at running them. I did some experiments closing database connections in finalizers and could generate any evidence that they were called at all. If and when finalizers are called is JVM implementation specific. So cross platform programming using finalizers is unpredictable at best, disastrous at worst.

The item also explains that uncaught exceptions thrown in finalizers are ignored, but the finalization of the object is terminated leaving it in an unknown and potentially corrupt state, which can result in arbitrary nondeterministic behavior. Using finalizers also increases the time to terminate an object by a whopping 430 times according to an (unexplained) test example run by Bloch.

The item describes the alternative to using finalizers as providing an explicit termination method that can, typically, be called from a finally block. This is what I do in a lot of cases and what the Java database objects provide.

Finalizers could be used as a safety net for forgotten terminate method invocations on the basis that it's better to release resource late than never, but of course there's no guarantee it'll get released at all as there is no guarantee that a finalizer will ever be called..

Item 65 – Don't Ignore Exceptions

The point that this item is trying to get across is that exceptions are trying to tell you that something bad has happened and should not be ignored. It's far too easy to write something like:

```
try
{
    ...
}
catch(Exception e)
{
}
```

The item points out that "an empty `catch` block defeats the purpose of exceptions, which is to force you to handle exceptional conditions" and "At the very least, the `catch` block should have a comment explaining why it is appropriate to ignore exceptions."

The item gives the example of closing a `FileInputStream` as a situation where it might be appropriate to ignore an exception, with a comment in the `catch` block or a message written to a log of course. The state of the file has not been changed and there is nothing roll back. This of course assumes that the file resource has been successfully release.

The advice in this item applies equally to checked and non-checked exceptions.

Sidebar 2: Effective Java Item 2: - Consider a builder when faced with many constructor parameters

The item gives an example of a class that has a number of optional properties that should be initialised via a constructor and explains how they can be initialized using the telescoping constructor pattern or the JavaBean pattern. It concludes that “the telescoping constructor pattern works, but it is hard to write client code where there are many parameters, and harder still to read it” and that “the JavaBean pattern precludes the possibility of making a class immutable.” His argument in both cases is convincing.

As a solution, the item suggests a variation on the builder pattern. Basically, the class with the optional properties has an inner class that can be used to initialise the properties on construction. The resulting initialisation syntax looks like this:

```
NutritionFacts cocaCola = new  
    NutritionFacts.Builder(240,8).calories(100).sodium(35).carbohydrate(27).build();
```

The item points out that “the builder pattern stimulates named optional parameters” and that the pattern does not really become useful until you have at least four optional properties and if needed should be used as soon as possible as refactoring to the pattern can be problematic.

The item summarises the builder pattern as “a good choice when designing classes whose constructors or static factories would have more than a handful of parameters.”

Acknowledgements

Thank you to Tony Barret-Powell, Kevlin Henney, Adrian Fagg and Russel Winder for advice and guidance (and for *not* spotting the huge flaw in my original proposal! ;-)).

References

[EffectiveJava] Effective Java: A Programming Language Guide by Joshua Bloch. ISBN-13: 978-0321356680

[IDisposable] C# IDisposable interface:
<http://msdn.microsoft.com/en-us/library/system.idisposable.aspx>

[HeiF] Handling Exceptions in Finally by Tony Barrett-Powell: <http://accu.org/index.php/journals/236>

[ExceptionalJava] Exceptional Java by Alan Griffiths: <http://accu.org/index.php/journals/399>

[MoreExceptionalJava] More Exceptional Java by Alan Griffiths:
<http://accu.org/index.php/journals/406>

[Generics] Java Generics: <http://java.sun.com/j2se/1.5.0/docs/guide/language/generics.html>

[JavaDocs] Sun Java Docs: <http://java.sun.com/reference/docs/>

[AToTP] Another Tail of Two Patterns:
<http://www.two-sdg.demon.co.uk/curbralan/papers/AnotherTaleOfTwoPatterns.pdf>

[JavaInANutshell] Java in a Nutshell by David Flanagan. ISBN-13: 978-0596007737